DATA STRUCTURES AND ALGORITHMS 3

O(n)

DSA3 2026-10-30

How the semester works, comprehensions,
and a feel for Big O

HOW THIS WORKS

What tutorials are for

Review

We take a deeper look at the week’s lecture topics, with fresh examples.

Project

A semester-long group project with checkpoints. More details later.

Preview?

Want a walkthrough of the new assignment? Just ask.

Stuck on something? Bring it. This isn’t a help desk, but questions are always welcome.

ASSIGNMENTS

Practice, on your own terms

0%

Not graded

Assignments don’t count toward your grade.

For practice

Each one exercises the ideas from the lectures. Doing them is strongly recommended.

Questions welcome

Stuck, curious, or want to compare approaches? Bring it to the tutorial.

NEW THIS YEAR

One project, all semester, as a group

WEIGHT	GROUP SIZE	CHECKPOINTS	FINAL DEADLINE
20%	3 people	About 5	TBD
of the course grade			After the final exam

Checkpoint meetings are mandatory

At each checkpoint, the whole group meets with us to discuss progress and direction.

GROUP PROJECT

Join the project group

Scan the code or open the link to give us your info.

forms.gle/cXu1u5S1MqoNLBt8A



SCAN TO SIGN UP

WORKING IN SPRINTS

Agile in one slide

THE OLD WAY

Plan it all, then build it all



Design, build, test, in that order. Problems surface at the very end, when they’re most expensive to fix.

AGILE

Build a little, check, adjust

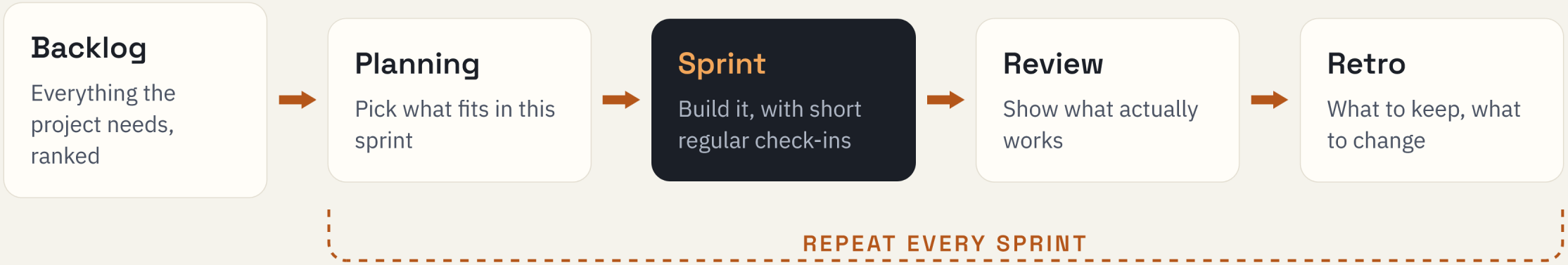


Short cycles that each end with something working. Feedback from one cycle shapes the plan for the next.

Working software early, and a plan that’s allowed to change.

WORKING IN SPRINTS

Scrum: one popular way to do agile



Sprints have a fixed length, usually 1 to 4 weeks.

Each one should end with something that runs.

Unfinished work goes back on the backlog.

ASSIGNMENT

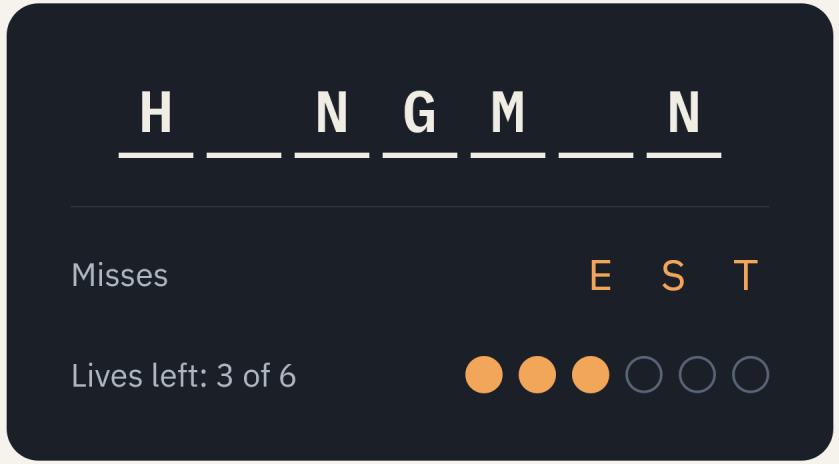
Hangman

Making smarter guesses with probabilities built from data

HANGMAN · REFRESHER

How the game works

1. A secret word is shown as one blank per letter
2. The guesser picks one letter at a time
3. **Hit:** the letter is revealed everywhere it appears
4. **Miss:** lose a life. Solve the word before they run out



In the assignment, **your code is the guesser**.

GUESS 1 OF 3

Which letter would you guess next?

Q _ _ _ _ _

Why?

GUESS 2 OF 3

Which letter would you guess next?

W A L K I _ _

Why?

GUESS 3 OF 3

Which letter would you guess next?

_ _ _ _ _

Why?

PROBABILITY FROM DATA

Your best guess depends on context

Q _ _ _ _	U	Q is almost always followed by U
W A L K I _ _	N	-ING is a very common ending
_ _ _ _ _	E	No clues, so fall back on overall letter frequency

Every guess is a bet on a **probability distribution** over letters. The clues you have reshape that distribution.

PROBABILITY FROM DATA

But where do the probabilities come from?



You get a **word list** with the assignment. What could it tell you?



How does what you **already know** about the word change that?



How would you turn that into **one guess**?

You just did this in your head. The assignment asks you to make it precise.

PYTHON

Comprehensions

Building lists, dicts and sets in a single expression

[x for x]

COMPREHENSIONS

A whole loop in one expression

```
[ x * 2 for x in nums if x > 0 ]
```

what to produce — where items come from — which to keep (optional)

Same result as this loop:

```
result = []
for x in nums:
    if x > 0:
        result.append(x * 2)
```

COMPREHENSIONS

Four shapes you'll write constantly

```
words = ["tree", "graph", "heap", "stack"]
```

TRANSFORM

```
[w.upper() for w in words]
# ['TREE', 'GRAPH', 'HEAP', 'STACK']
```

FILTER

```
[w for w in words if len(w) > 4]
# ['graph', 'stack']
```

BOTH AT ONCE

```
[w[0] for w in words if "a" in w]
# ['g', 'h', 's']
```

OVER A STRING

```
[c for c in "banana" if c != "a"]
# ['b', 'n', 'n']
```

COMPREHENSIONS

Where the if goes changes everything

```
nums = [1, 2, 3, 4]
```

At the end: a filter

```
[x * 2 for x in nums if x % 2 == 0]
# [4, 8]
```

Drops items. The result can be shorter than the input. No **else** allowed.

At the front: a choice

```
[x * 2 if x % 2 == 0 else x for x in nums]
# [1, 4, 3, 8]
```

Keeps every item and picks a value for each. The **else** is required.

COMPREHENSIONS

Same idea, other containers

```
words = ["tree", "graph", "heap", "stack"]
```

Dict: {key: value for ...}

```
{w: len(w) for w in words}
# {'tree': 4, 'graph': 5, ...}
```

```
prices = {"tea": 3, "cake": 6, "bun": 2}
{k: v for k, v in prices.items() if v < 5}
# {'tea': 3, 'bun': 2}
```

Set: {value for ...}

```
{len(w) for w in words}
# {4, 5}  duplicates vanish
```

```
{x % 3 for x in range(10)}
# {0, 1, 2}
```

⚠ **Gotcha:** a bare {} is an empty dict. An empty set is set().

COMPLEXITY

Big O is a feeling before it's a formula

Everyone needs the intuition

Spot the slow loop, pick the right data structure, predict whether code survives real input.

Some need the rigor

Algorithm research, competitive programming, performance-critical systems.

COMPLEXITY · WARM-UP

Feel the growth rate

A O(?)

```
def largest(nums):
    best = nums[0]
    for x in nums:
        if x > best:
            best = x
    return best
```

B O(?)

```
def center(nums):
    mean = sum(nums) / len(nums)
    return [x - mean for x in nums]
```

C O(?)

```
def has_duplicate(nums):
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[i] == nums[j]:
                return True
    return False
```

D O(?)

```
def running_checks(a, b):
    hits = 0
    for i in range(len(a)):
        if b[i] == sum(a[:i + 1]):
            hits += 1
    return hits
```

COMPLEXITY · WARM-UP

What your gut should have said

	Growth	Why
A	O(n)	One pass over the input
B	O(n)	Two passes back to back is 2n steps; constants drop
C	O(n²)	Every pair, about n²/2 comparisons in the worst case
D	O(n²)	One visible loop, but sum(a[:i + 1]) is a hidden loop costing O(i)

Count the loops you *can't* see: slices, in on a list, sum, sorted, string concatenation.

COMPLEXITY · ROUND 2

Feel the growth rate, again

A O(?)

```
def halving_steps(nums):
    steps = 0
    for x in nums:
        size = len(nums)
        while size > 1:
            size = size // 2
            steps += 1
    return steps
```

B O(?)

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

C O(?)

```
def digit_count(n):
    count = 1
    while n >= 10:
        n //= 10
        count += 1
    return count
```

D O(?)

```
def count_pair_sums(nums, target):
    count = 0
    for x in nums:
        for y in nums:
            if x + y == target:
                count += 1
    return count
```

COMPLEXITY · ROUND 2

What your gut should have said

	Growth	Why
A	O(n log n)	n passes, and each one halves from n down to 1: log n steps
B	$O(2^n)$	Every call makes two more: the call tree roughly doubles per level
C	O(log n)	Each step divides n by 10; the base of the log doesn't matter
D	$O(n^2)$	Every x is checked against every y: n × n checks

Halving: log n. Halving, n times: n log n. All pairs: n^2. Branching twice per call: 2^n.

COMPLEXITY

The usual suspects

Class	Name	The shape you'll recognize
O(1)	Constant	Index into a list, average dict or set lookup
O(log n)	Logarithmic	Halve the problem each step: binary search
O(n)	Linear	One pass over the input
O(n log n)	Linearithmic	Good comparison sorts: sorted(), merge sort
$O(n^2)$	Quadratic	All pairs: nested loops over the input
$O(2^n)$	Exponential	Every subset; naive recursive Fibonacci
O(n!)	Factorial	Every ordering; brute-force route planning

COMPLEXITY

How fast “fast-growing” really is

Steps	n = 10	n = 100	n = 1,000
$\log_2 n$	3.3	6.6	10
n	10	100	1,000
$n \log_2 n$	33	664	≈10,000
n^2	100	10,000	1,000,000
2^n	1,024	$\approx 1.3 \times 10^{30}$	$\approx 1.1 \times 10^{301}$
$n!$	3,628,800	$\approx 9.3 \times 10^{157}$	$\approx 4 \times 10^{2567}$

At a billion steps per second, 2^n with n = 100 runs thousands of times longer than the universe has existed.

COMPLEXITY · EXPONENTIAL

Same answer, wildly different cost

RECURSIVE — O(2ⁿ)

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

ITERATIVE — O(n)

```
def fib(n):
    prev, cur = 0, 1
    for _ in range(n):
        prev, cur = cur, prev + cur
    return prev
```

Try it: time fib(35) both ways with time.perf_counter(). Then put @functools.cache on the recursive one and time it again.

COMPLEXITY · FACTORIAL

Trying every ordering costs n!

```
def orderings(items):
    if not items:
        return [[]]
    result = []
    for i, first in enumerate(items):
        rest = items[:i] + items[i + 1:]
        for p in orderings(rest):
            result.append([first] + p)
    return result
```

Brute-force round trips through every city

181,440

distinct tours for 10 cities

≈6 × 10¹⁶

distinct tours for 20 cities

COMPLEXITY

Is Big O the whole story?

Two functions solve the same problem: one is O(n log n), the other O(n²). You time them, and on small inputs the O(n²) one wins. What's going on?

Constants hide

Big O drops them. 50·n log n loses to 2·n² until n reaches the hundreds.

Overhead is real

Function calls, recursion and allocation all cost time on every step.

Setup isn't free

A set makes lookups O(1), but building it costs a full pass first. For one lookup, a list is fine.

Big O says who wins eventually. Measuring says who wins at your n.

FUNCTIONS THAT CALL THEMSELVES

Recursion

When it’s the natural fit, and what it really costs

RECURSION

When it fits, and what it costs

Fits naturally

- Tree and graph traversals
- Divide and conquer: merge sort, binary search
- Self-similar structures: fractals, nested data
- Definitions that are already recursive

Costs to watch

- Every call adds a stack frame: memory and time
- Python stops at about 1,000 nested calls by default
- No tail-call optimization in Python
- Repeated subproblems explode unless you cache

EXERCISE · ANY LANGUAGE

You have a list of n numbers. Remove the duplicates.

How many different approaches can you find?

What does each one cost in time and memory?

Does the original order survive?

EXERCISE · SOME ANSWERS

Four ways to remove duplicates

1 · Check what you kept

```
result = []
for x in nums:
    if x not in result:
        result.append(x)
```

O(n²) time · no extra structure · order kept

2 · Remember what you've seen

```
seen, result = set(), []
for x in nums:
    if x not in seen:
        seen.add(x)
        result.append(x)
```

O(n) time · an extra set · order kept

3 · Let a set do it

```
result = list(set(nums))
```

O(n) time · an extra set · **order lost**

4 · Let a dict do it

```
result = list(dict.fromkeys(nums))
```

O(n) time · an extra dict · order kept

The difference between 1 and 2 is one data structure: **spend memory, save time.**

Questions?

O(1)